Perl Short Course: Some Extra Notes

Duncan C. White (d.white@imperial.ac.uk)

Dept of Computing,
Imperial College London

December 2012

- Here's some material that got removed from the Perl course, might be interesting:
  - **Fifth Session: DBI films example, handling null/undef better**
  - **Sixth Session: Person and Programmer version 4**
  - **How CSG use Perl**

- In the DBI films example eg6, slide 19, we found a problem: one film record had a null **length** field (which became **undef** in Perl). This caused a warning, which we fixed in the lecture by:

```
while( my $record = $sth->fetchrow_hashref )
{
  print "Title:    $record->{title}\n";
  print "Director: $record->{director}\n";
  print "Origin:   $record->{origin}\n";
  print "Made:     $record->{made}\n";
  my $length = $record->{length} // '';
  print "Length:   $length\n";
  print "-" x 30 . "\n";
}
$sth->finish;
```

- Perhaps *any of the fields* could have been null, could we deal with all of them better? Sounds like a job for `map`!

```
while( my $record = $sth->fetchrow_hashref )
{
  my( $title, $director, $origin, $made, $length ) =
      map { $record->{$_} // '' }
          qw(title director origin made length);
  print "Title:    $title\n";
  print "Director: $director\n";
  print "Origin:   $origin\n";
  print "Made:     $made\n";
  print "Length:   $length\n";
  print "-" x 30 . "\n";
}
$sth->finish;
```

- Or, if you preferred, use `map` to generate a new hash:

```
while( my $record = $sth->fetchrow_hashref )
{
  my %f = map { $_ => $record->{$_} // '' } keys %$record;
  print "Title:    $f{title}\n";
  print "Director: $f{director}\n";
  print "Origin:   $f{origin}\n";
  print "Made:     $f{made}\n";
  print "Length:   $f{length}\n";
  print "-" x 30 . "\n";
}
$sth->finish;
```

- Or we could use a procedural map to modify `%$record`:

```
while( my $record = $sth->fetchrow_hashref )
{
  map { $record->{$_} //= '' } keys %record;
  print "Title:    $title\n";
  print "Director: $director\n";
  print "Origin:   $origin\n";
  print "Made:     $made\n";
  print "Length:   $length\n";
  print "-" x 30 . "\n";
}
$sth->finish;
```

- In the sixth lecture, we presented a lengthy class example about people (class `Person`) with names, sexes and ages, and programmers (subclass `Programmer`) with additional programming language skills.
- Our final version *programmer-v3* got Duncan's skills back by using *constructor chaining*.

- In the sixth lecture, we presented a lengthy class example about people (class `Person`) with names, sexes and ages, and programmers (subclass `Programmer`) with additional programming language skills.
- Our final version *programmer-v3* got Duncan's skills back by using *constructor chaining*.
- Isn't there a better way? Well, the only thing varying per-class appears to be the set of data fields which we want to initialize, and their default values. Remove `Programmer's` constructor, and generalise `Person's` constructor as follows:

```
fun new( $class, %arg ) {
  my $obj     = bless( {}, $class );
  my %default = $obj->_defaultvalues;
  while( my($datum,$value) = each(%default) )
  {
    $obj->{$datum} = $arg{$datum} // $value;
  }
  return $obj;
}
```

- In the sixth lecture, we presented a lengthy class example about people (class `Person`) with names, sexes and ages, and programmers (subclass `Programmer`) with additional programming language skills.
- Our final version *programmer-v3* got Duncan's skills back by using *constructor chaining*.
- Isn't there a better way? Well, the only thing varying per-class appears to be the set of data fields which we want to initialize, and their default values. Remove `Programmer's` constructor, and generalise `Person's` constructor as follows:

```
fun new( $class, %arg ) {
  my $obj     = bless( {}, $class );
  my %default = $obj->_defaultvalues;
  while( my($datum,$value) = each(%default) )
  {
    $obj->{$datum} = $arg{$datum} // $value;
  }
  return $obj;
}
```

- Now, each class defines a private `_defaultvalues` method, listing the default values of all the initializable data fields:

```
method Person::_defaultvalues { return (NAME=>"Shirley", SEX=>"f", AGE=>26); }
```

- Continuing:

```
method Programmer::_defaultvalues
{
  return ( NAME=>"Shirley", SEX=>"f", AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- Continuing:

```
method Programmer::_defaultvalues
{
  return ( NAME=>"Shirley", SEX=>"f", AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- These methods allow a single generic `Person::new` constructor to initialize all the desired data fields. Of course, we are still repeating all the defaults in each subclass.

- Continuing:

```
method Programmer::_defaultvalues
{
  return ( NAME=>"Shirley", SEX=>"f", AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- These methods allow a single generic `Person::new` constructor to initialize all the desired data fields. Of course, we are still repeating all the defaults in each subclass.
- Can we fix this? Yes, with method chaining!

```
method Programmer::_defaultvalues {
  my %default = $self->Person::_defaultvalues;
  $default{SKILLS} = { java => "ok" };
  return %default;
}
```

- Continuing:

```
method Programmer::_defaultvalues
{
  return ( NAME=>"Shirley", SEX=>"f", AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- These methods allow a single generic `Person::new` constructor to initialize all the desired data fields. Of course, we are still repeating all the defaults in each subclass.
- Can we fix this? Yes, with method chaining!

```
method Programmer::_defaultvalues {
  my %default = $self->Person::_defaultvalues;
  $default{SKILLS} = { java => "ok" };
  return %default;
}
```

- More generically, we can write the chained method call as:

```
 my %default = $self->SUPER::_defaultvalues;
```

  to call the first available parental `_defaultvalues` method.

- Continuing:

```
method Programmer::_defaultvalues
{
  return ( NAME=>"Shirley", SEX=>"f", AGE=>26, SKILLS=>{java=>"ok"} );
}
```

- These methods allow a single generic `Person::new` constructor to initialize all the desired data fields. Of course, we are still repeating all the defaults in each subclass.
- Can we fix this? Yes, with method chaining!

```
method Programmer::_defaultvalues {
  my %default = $self->Person::_defaultvalues;
  $default{SKILLS} = { java => "ok" };
  return %default;
}
```

- More generically, we can write the chained method call as:

```
 my %default = $self->SUPER::_defaultvalues;
```

  to call the first available parental `_defaultvalues` method.
- This is the final version `programmer-v4` in the Lecture 6 tarball.

I made some off the cuff remarks about how CSG use Perl. Let's summarise them quickly:

- When we unpack a new machine, we enter inventory, hostname, IP address and MAC address details into a Postgres database - via a **web database interface (a Perl CGI script)**. We also add configuration information about the new desktop in another database table (host classes, what type of machine it is) and do a small amount of extra configuration.

I made some off the cuff remarks about how CSG use Perl. Let's summarise them quickly:

- When we unpack a new machine, we enter inventory, hostname, IP address and MAC address details into a Postgres database - via a **web database interface (a Perl CGI script)**. We also add configuration information about the new desktop in another database table (host classes, what type of machine it is) and do a small amount of extra configuration.
- When we plug the new machine into the network and turn it on, we boot from a Linux USB key which NFS mounts a "root filesystem" and then starts running a **CSG-custom installation and maintenance system, entirely written in Perl**.
  - First, this chooses which disk(s) to use as the boot disk.
  - Then it partitions those disks using whichever partitioning scheme the machine's host class info indicates should be used (the partitioning scheme is written in Perl too), optionally creates Linux logical volumes on the partitions, creates filesystems on those partitions/logical volumes, and mounts them.

- Continuing:
  - Then it replicates the NFS root filesystem to the fresh root filesystem, and switches into it.
  - Then successive maintenance scripts copy in numerous configuration files (universal to DoC, or specific to the machine's host classes), and install hundreds (servers) or thousands (desktops) of packages - selecting the appropriate package list for that type of machine (via host classes).
  - Finally, the machine boots, and on first proper boot is a full member of DoC, running the right services, having the right packages, knowing about DoC users etc etc.
  - The maintenance system regularly runs thereafter, keeping the machine up to date, installing new kernels, new packages, tweaking config files when we decide they need tweaking, etc etc.

- Continuing:
  - Then it replicates the NFS root filesystem to the fresh root filesystem, and switches into it.
  - Then successive maintenance scripts copy in numerous configuration files (universal to DoC, or specific to the machine's host classes), and install hundreds (servers) or thousands (desktops) of packages - selecting the appropriate package list for that type of machine (via host classes).
  - Finally, the machine boots, and on first proper boot is a full member of DoC, running the right services, having the right packages, knowing about DoC users etc etc.
  - The maintenance system regularly runs thereafter, keeping the machine up to date, installing new kernels, new packages, tweaking config files when we decide they need tweaking, etc etc.
- **Lexis, our locally developed exam lockdown system**, is entirely written in Perl - the client code that chats the Lexis custom protocol, the Lexis server (that uses Perl threads!), and a separate Perl/Tk status monitor.

- Continuing:
  - Then it replicates the NFS root filesystem to the fresh root filesystem, and switches into it.
  - Then successive maintenance scripts copy in numerous configuration files (universal to DoC, or specific to the machine's host classes), and install hundreds (servers) or thousands (desktops) of packages - selecting the appropriate package list for that type of machine (via host classes).
  - Finally, the machine boots, and on first proper boot is a full member of DoC, running the right services, having the right packages, knowing about DoC users etc etc.
  - The maintenance system regularly runs thereafter, keeping the machine up to date, installing new kernels, new packages, tweaking config files when we decide they need tweaking, etc etc.
- **Lexis, our locally developed exam lockdown system**, is entirely written in Perl - the client code that chats the Lexis custom protocol, the Lexis server (that uses Perl threads!), and a separate Perl/Tk status monitor.
- Plus a million Perl "helper" scripts, Perl one liners.
- Conclusion: **Perl code runs DoC**